Software Design, Modelling and Analysis in UML

Lecture 14: Hierarchical State Machines I

2012-12-19

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:
- RTC-Rules: Discard, Dispatch, Commence.

This Lecture:
- Educational Objectives: Capabilities for following tasks/questions.
  - What does this State Machine mean? What happens if I inject this event?
  - Can you please model the following behaviour.
  - What is: initial state.
  - What does this hierarchical State Machine mean? What may happen if I inject this event?
  - What is: AND-State, OR-State, pseudo-state, entry/exit/do, final state, ..

- Content:
  - Step, RTC, Divergence
  - Putting It All Together
  - Rhapsody Demo
  - Hierarchical State Machines Syntax
Step and Run-to-completion Step

Notions of Steps: The Step

**Note:** we call one evolution $(\sigma, \varepsilon) \xrightarrow{\text{cons}, \text{Snd}, u} (\sigma', \varepsilon')$ a step.

Thus in our setting, a step directly corresponds to

one object (namely $u$) takes a single transition between regular states.

(We have to extend the concept of “single transition” for hierarchical state machines.)

**That is:** We’re going for an interleaving semantics without true parallelism.
Notions of Steps: The Run-to-Completion Step

What is a run-to-completion step...?

- **Intuition**: a maximal sequence of steps, where the first step is a dispatch step and all later steps are commence steps.

- **Note**: one step corresponds to one transition in the state machine.

A run-to-completion step is in general not syntactically definable — one transition may be taken multiple times during an RTC-step.

### Example:

![Diagram of a run-to-completion step]

- \( \sigma_1 : x = 2 \)
- \( \epsilon \) for \( u \)
- \( E \) for \( u \)
- \( E[x > 0] / x := x - 1 \)

Notions of Steps: The Run-to-Completion Step Cont’d

**Proposal**: Let

\[
(\sigma_0, \epsilon_0) \xrightarrow{\text{(cons}_0, \text{Snd}_0)} u_0 \xrightarrow{} \cdots \xrightarrow{} (\text{cons}_{n-1}, \text{Snd}_{n-1}) u_{n-1} \xrightarrow{} (\sigma_n, \epsilon_n), \quad n > 0,
\]

be a finite (!), non-empty, maximal, consecutive sequence such that

- object \( u \) is alive in \( \sigma_0 \),
- \( u_0 = u \) and \( (\text{cons}_0, \text{Snd}_0) \) indicates dispatching to \( u \), i.e. \( \text{cons}_0 = \{(u, \vec{v} \mapsto \vec{d})\} \),
- there are no receptions by \( u \) in between, i.e.

\[
\text{cons}_i \cap \{u\} \times \text{Evs}(\epsilon, \emptyset) = \emptyset, \quad i > 1,
\]

- \( u_{n-1} = u \) and \( u \) is stable only in \( \sigma_0 \) and \( \sigma_n \), i.e.

\[
\sigma_0(u)(\text{stable}) = \sigma_n(u)(\text{stable}) = 1 \text{ and } \sigma_i(u)(\text{stable}) = 0 \text{ for } 0 < i < n.
\]

Let \( 0 = k_1 < k_2 < \cdots < k_N = n \) be the maximal sequence of indices such that \( u_{k_i} = u \) for \( 1 \leq i \leq N \). Then we call the sequence

\[
(\sigma_0(u) = \sigma_{k_1}(u), \sigma_{k_2}(u), \ldots, \sigma_{k_N}(u) = \sigma_{n-1}(u))
\]

a (!) **run-to-completion computation** of \( u \) (from (local) configuration \( \sigma_0(u) \)).
Divergence

We say, object \( u \) can diverge on reception \( \textit{cons} \) from (local) configuration \( \sigma_0(u) \) if and only if there is an infinite, consecutive sequence

\[
(\sigma_0, \varepsilon_0) \xrightarrow{\text{(cons, Snd}_{u})} (\sigma_1, \varepsilon_1) \xrightarrow{\text{(cons, Snd}_{u})} \ldots
\]

such that \( u \) doesn’t become stable again.

- **Note**: disappearance of object not considered in the definitions.

By the current definitions, it’s not divergence but an RTC-step.

Run-to-Completion Step: Discussion.

What people may dislike on our definition of RTC-step is that it takes a global and non-compositional view. That is:

- In the projection onto a single object we still see the effect of interaction with other objects.
- Adding classes (or even objects) may change the divergence behaviour of existing ones.
- Compositional would be: the behaviour of a set of objects is determined by the behaviour of each object “in isolation”.

Our semantics and notion of RTC-step doesn’t have this (often desired) property.

Can we give (syntactical) criteria such that any global run-to-completion step is an interleaving of local ones?

**Maybe**: Strict interfaces.  
(Proof left as exercise...)

- (A): Refer to private features only via “self”.
  (Recall that other objects of the same class can modify private attributes.)
- (B): Let objects only communicate by events, i.e. don’t let them modify each other’s local state via links at all.
The Missing Piece: Initial States

Recall: a labelled transition system is $(S,\rightarrow, S_0)$. We have

- $S$: system configurations $(\sigma, \varepsilon)$
- $\rightarrow$: labelled transition relation $(\sigma, \varepsilon) \xrightarrow{(\text{cons}, \text{Snd})} (\sigma', \varepsilon')$.

Wanted: initial states $S_0$.

Proposal: Require a (finite) set of object diagrams $\mathcal{OD}$ as part of a UML model $(\mathcal{E} \mathcal{D}, \mathcal{M}, \mathcal{O} \mathcal{D})$.

And set

$$S_0 = \{ (\sigma,\varepsilon) \mid \sigma \in G^{-1}(\mathcal{OD}), \mathcal{OD} \in \mathcal{O} \mathcal{D}, \varepsilon \text{ empty} \}.$$ 

Other Approach: (used by Rhapsody tool) multiplicity of classes.

We can read that as an abbreviation for an object diagram.
Semantics of UML Model — So Far

The semantics of the UML model

\[ M = (C_2, I, O) \]

where

- some classes in \( C_2 \) are stereotyped as 'signal' (standard), some signals and attributes are stereotyped as 'external' (non-standard),
- there is a 1-to-1 relation between classes and state machines,
- \( O \) is a set of object diagrams over \( C_2 \),

is the transition system \( (S, \rightarrow, S_0) \) constructed on the previous slide.

The computations of \( M \) are the computations of \( (S, \rightarrow, S_0) \) given by \( \sigma \).

OCL Constraints and Behaviour

- Let \( M = (C_2, I, O) \) be a UML model.
- We call \( M \) consistent iff, for each OCL constraint \( expr \in Inv(C_2) \),
  \( \sigma \models expr \) for each "reasonable point" \( (\sigma, \varepsilon) \) of computations of \( M \).

Note: we could define \( Inv(\mathcal{M}) \) similar to \( Inv(C_2) \).

Pragmatics:

- In UML-as-blueprint mode, if \( \mathcal{M} \) doesn’t exist yet, then \( M = (C_2, \emptyset, O) \) is typically asking the developer to provide \( \mathcal{M} \) such that \( M' = (C_2, \mathcal{M}, O) \) is consistent.
  - If the developer makes a mistake, then \( M' \) is inconsistent.
- Not common: if \( \mathcal{M} \) is given, then constraints are also considered when choosing transitions in the RTC-algorithm. In other words: even in presence of mistakes, the \( \mathcal{M} \) never move to inconsistent configurations.
Hierarchical State Machines

UML State-Machines: What do we have to cover?
The Full Story

UML distinguishes the following kinds of states:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple state</td>
<td><img src="image1.png" alt="Example" /></td>
</tr>
<tr>
<td>final state</td>
<td><img src="image2.png" alt="Example" /></td>
</tr>
<tr>
<td>composite state</td>
<td><img src="image3.png" alt="Example" /></td>
</tr>
<tr>
<td>OR</td>
<td><img src="image4.png" alt="Example" /></td>
</tr>
<tr>
<td>AND</td>
<td><img src="image5.png" alt="Example" /></td>
</tr>
<tr>
<td>pseudo-state</td>
<td><img src="image6.png" alt="Example" /></td>
</tr>
<tr>
<td>initial</td>
<td><img src="image7.png" alt="Example" /></td>
</tr>
<tr>
<td>(shallow) history</td>
<td><img src="image8.png" alt="Example" /></td>
</tr>
<tr>
<td>deep history</td>
<td><img src="image9.png" alt="Example" /></td>
</tr>
<tr>
<td>fork/join</td>
<td><img src="image10.png" alt="Example" /></td>
</tr>
<tr>
<td>junction, choice</td>
<td><img src="image11.png" alt="Example" /></td>
</tr>
<tr>
<td>entry point</td>
<td><img src="image12.png" alt="Example" /></td>
</tr>
<tr>
<td>exit point</td>
<td><img src="image13.png" alt="Example" /></td>
</tr>
<tr>
<td>terminate</td>
<td><img src="image14.png" alt="Example" /></td>
</tr>
<tr>
<td>submachine state</td>
<td><img src="image15.png" alt="Example" /></td>
</tr>
</tbody>
</table>

Representing All Kinds of States

- Until now:

\[(S, s_0, \rightarrow), \quad s_0 \in S, \rightarrow \subseteq S \times (\mathcal{E} \cup \{\bot\}) \times Expr_{\mathcal{S}} \times Act_{\mathcal{S}} \times S\]
Representing All Kinds of States

- Until now:
  \[(S, s_0, \rightarrow), \quad s_0 \in S, \rightarrow \subseteq S \times (\mathcal{E} \cup \{\_\}) \times \text{Expr}_{\mathcal{F}} \times \text{Act}_{\mathcal{F}} \times S\]

- From now on: (hierarchical) state machines
  \[(S, \text{kind}, \text{region}, \rightarrow, \psi, \text{annot})\]

  where
  - \(S \supseteq \{\text{top}\}\) is a finite set of states (as before),
  - \(\text{kind} : S \rightarrow \{\text{st}, \text{init}, \text{fin}, \text{shist}, \text{dhist}, \text{fork}, \text{join}, \text{junc}, \text{choi}, \text{ent}, \text{exi}, \text{term}\}\) is a function which labels states with their kind, (new)
  - \(\text{region} : S \rightarrow 2^{2^S}\) is a function which characterises the regions of a state, (new)
  - \(\rightarrow\) is a set of transitions, (changed)
  - \(\psi : (\rightarrow) \rightarrow 2^S \times 2^S\) is an incidence function, and (new)
  - \(\text{annot} : (\rightarrow) \rightarrow (\mathcal{E} \cup \{\_\}) \times \text{Expr}_{\mathcal{F}} \times \text{Act}_{\mathcal{F}}\) provides an annotation for each transition. (new)

(s_0 is then redundant — replaced by proper state (!) of kind ‘init’.)
(S, kind, region, \rightarrow, \psi, annot)

### simple state

- \text{example} ∈ S
- \text{kind} = \text{st}
- \text{region} = \emptyset

### final state

- \text{example} ∈ S
- \text{kind} = \text{fin}
- \text{region} = \emptyset

### composite state

<table>
<thead>
<tr>
<th>OR</th>
<th>\text{example} ∈ S</th>
<th>\text{kind}</th>
<th>\text{region}</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>\text{example} ∈ S</td>
<td>\text{kind}</td>
<td>\text{region}</td>
</tr>
</tbody>
</table>

### submachine state

- \text{example} ∈ S
- \text{kind} = \text{(later)}
- \text{region} = \emptyset

### pseudo-state

- \text{example} ∈ S
- \text{kind} = \text{(s,kind(s)) for short}
- \text{region} = \emptyset

DON’T! DON’T!

\text{don’t!} [gd] / \text{act}

\text{annot}

... translates to (S, kind, region, \rightarrow, \psi, annot) =

\[
\left(\begin{array}{c}
\{ \text{top}, \text{st} \}, \{ \text{x1}, \text{init} \}, \{ \text{x2}, \text{st} \}, \{ \text{x2}, \text{fin} \} \\
\{ \text{x1}, \text{x2} \} \\
\{ \text{t1},\text{t2} \} \\
\{ \text{t1} \rightarrow \{ \text{x1} \}, \text{t2} \rightarrow \{ \text{x2} \} \} \\
\end{array}\right)
\]
Well-Formedness: Regions (follows from diagram)

<table>
<thead>
<tr>
<th>∈ S</th>
<th>kind</th>
<th>region ⊆ 2^S, S_i ⊆ S</th>
<th>child ⊆ S</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple state</td>
<td>s</td>
<td>st</td>
<td>∅</td>
</tr>
<tr>
<td>final state</td>
<td>s</td>
<td>fin</td>
<td>∅</td>
</tr>
<tr>
<td>composite state</td>
<td>s</td>
<td>st</td>
<td>{S_1, \ldots, S_n}, n ≥ 1</td>
</tr>
<tr>
<td>pseudo-state</td>
<td>s</td>
<td>init, \ldots</td>
<td>∅</td>
</tr>
<tr>
<td>implicit top state</td>
<td>top</td>
<td>st</td>
<td>{S_1}</td>
</tr>
</tbody>
</table>

- Each state (except for top) lies in exactly one region,
- States s ∈ S with \( \text{kind}(s) = \text{st} \) may comprise regions.
  - No region: simple state.
  - One region: OR-state.
  - Two or more regions: AND-state.
- Final and pseudo states don’t comprise regions.
- The region function induces a child function.

Well-Formedness: Initial State (requirement on diagram)

- Each non-empty region has a reasonable initial state and at least one transition from there, i.e.
  - for each s ∈ S with \( \text{region}(s) = \{S_1, \ldots, S_n\}, \ n ≥ 1 \), for each 1 ≤ i ≤ n,
  - there exists exactly one initial pseudo-state \( (s_i^1, \text{init}) \in S_i \) and at least one transition t ∈→ with \( s_i^1 \) as source,
  - and such transition’s target \( s_i^2 \) is in \( S_i \), and (for simplicity) \( \text{kind}(s_i^2) = \text{st} \), and \( \text{annot}(t) = (\_ \_ \text{true}, \_ \_ \text{act}) \).
- No ingoing transitions to initial states.
- No outgoing transitions from final states.

- Recall:

\[ \text{DON’T!} \quad \text{DON’T!} \]

\[ \text{tr}[\text{gd}]/\text{act} \]

\[ s \quad \text{annot} \]
- Initial pseudostate, final state.
- Composite states.
- Entry/do/exit actions, internal transitions.
- History and other pseudostates, the rest.

*Initial Pseudostates and Final States*
**Initial Pseudostate**

**Principle:**
- when entering a region **without** a specific destination state,
- then go to a state which is destination of an initiation transition,
- execute the action of the chosen initiation transitions **between** exit and entry actions.

**Special case:** the region of **top**.
- If class \( C \) has a state-machine, then “create-\( C \) transformer” is the concatenation of
  - the transformer of the “constructor” of \( C \) (here not introduced explicitly) and
  - a transformer corresponding to one initiation transition of the top region.
Towards Final States: Completion of States

- Transitions without trigger can **conceptionally** be viewed as being sensitive for the “completion event”.

- Dispatching (here: $E$) can **then alternatively** be viewed as

(i) fetch event (here: $E$) from the ether,
Towards Final States: Completion of States

Transitions without trigger can \textit{conceptionally} be viewed as being sensitive for the “completion event”.

Dispatching (here: $E$) \textit{can then alternatively} be \textit{viewed} as

(i) fetch event (here: $E$) from the ether,
(ii) take an enabled transition (here: to $s_2$),
(iii) remove event from the ether,
Towards Final States: Completion of States

- Transitions without trigger can *conceptionally* be viewed as being sensitive for the “completion event”.

- Dispatching (here: E) *can then alternatively* be *viewed* as
  
  (i) fetch event (here: E) from the ether,
  
  (ii) take an enabled transition (here: to s₂),
  
  (iii) remove event from the ether,
  
  (iv) after having finished entry and do action of current state (here: s₂) — the state is then called *completed* —,

  (v) raise a *completion event* — with strict priority over events from ether!
Towards Final States: Completion of States

- Transitions without trigger can **conceptionally** be viewed as being sensitive for the “completion event”.
- Dispatching (here: $E$) can then **alternatively** be **viewed** as
  (i) fetch event (here: $E$) from the ether,
  (ii) take an enabled transition (here: to $s_2$),
  (iii) remove event from the ether,
  (iv) after having finished entry and do action of current state (here: $s_2$) — the state is then called **completed** —,
  (v) raise a **completion event** — with strict priority over events from ether!
  (vi) if there is a transition enabled which is sensitive for the completion event,
      • then take it (here: $(s_2, s_3)$),
      • otherwise become stable.

Final States

- If
  • a step of object $u$ moves $u$ into a final state ($s, fin$), and
  • all sibling regions are in a final state,
then (conceptionally) a completion event for the current composite state $s$ is raised.
Final States

• If
  • a step of object \( u \) moves \( u \) into a final state \((s, \text{fin})\), and
  • all sibling regions are in a final state,
then (conceptionally) a completion event for the current composite state \( s \) is raised.

• If there is a transition of a parent state (i.e., inverse of child) of \( s \) enabled which is sensitive for the completion event,
  • then take that transition,
  • otherwise kill \( u \)
\( \rightsquigarrow \) adjust (2.) and (3.) in the semantics accordingly

• One consequence: \( u \) never survives reaching a state \((s, \text{fin})\) with \( s \in \text{child}(\text{top}) \).
Final States

- If
  - a step of object \( u \) moves \( u \) into a final state \((s, \text{fin})\), and
  - all sibling regions are in a final state,
then (conceptionally) a completion event for the current composite state \( s \) is raised.
- If there is a transition of a parent state (i.e., inverse of \( \text{child} \)) of \( s \) enabled which is sensitive for the completion event,
  - then take that transition,
  - otherwise kill \( u \)
\( \rightsquigarrow \) adjust (2.) and (3.) in the semantics accordingly

- One consequence: \( u \) never survives reaching a state \((s, \text{fin})\) with \( s \in \text{child}(\text{top}) \).

- Now: in Core State Machines, there is no parent state.
- Later: in Hierarchical ones, there may be one.

---

Composite States
(formalisation follows [?])
Composite States

- In a sense, composite states are about **abbreviation, structuring**, and **avoiding redundancy**.

- Idea: in Tron, for the Player’s Statemachine, instead of

```plaintext
write
```

![Diagram of Composite States](image)

resigned

```plaintext
X/
```

```
X/
```

```
X/
```

```
X/
```

```
X/
```

```
resigned
```

29:16
Composite States

and instead of

write
Recall: Syntax

\[
\begin{align*}
&\text{translates to} \\
&\{(\{\text{top, st}\}, \{s, st\}, \{s_1, st\}, \{s_2, st\}, \{s_2, st\}, \{s_3, st\}, \{s_3, st\})\}, \\
&\{\text{top} \mapsto \{s\}, s \mapsto \{\{s_1, s'_1\}, \{s_2, s'_2\}, \{s_3, s'_3\}\}, s_1 \mapsto \emptyset, s'_1 \mapsto \emptyset, \ldots\}, \\
&\to, \psi, \text{annot}\}
\end{align*}
\]

Syntax: Fork/Join

• For brevity, we always consider transitions with (possibly) multiple sources and targets, i.e.
  \[\psi : (\to) \to (2^S \setminus \emptyset) \times (2^S \setminus \emptyset)\]

• For instance,

\[
\begin{align*}
&\text{translates to} \\
&(S, \text{kind, region}, \{t_1\}, \{t_1 \mapsto \{\{s_2, s_3\}, \{s_3, s_6\}\}\}, \{t_1 \mapsto (\text{tr, gd, act})\})
\end{align*}
\]

• Naming convention: \(\psi(t) = (\text{source}(t), \text{target}(t))\).
Composite States: Blessing or Curse?

States:
- what are legal state configurations?
- what is the type of the implicit \( s \) attribute?

Transitions:
- what are legal transitions?
- when is a transition enabled?
- what effects do transitions have?

- what may happen on \( E \)?
- what may happen on \( E, F \)?
- can \( E, G \) kill the object?
- ...

- what may happen on \( E \)?
- what may happen on \( E, F \)?
- can \( E, G \) kill the object?
- ...

33/66
State Configuration

- The type of $st$ is from now on a set of states, i.e. $st : 2^S$
- A set $S_1 \subseteq S$ is called (legal) state configurations if and only if
  - $top \in S_1$, and
  - with each state $s \in S_1$ that has a non-empty region $\emptyset \neq R \in region(s)$, exactly one (non pseudo-state) child of $s$ is in $S_1$, i.e.
    \[ |\{s \in R | \text{kind}(s) \in \{\text{st, fin}\}\} \cap S_1| = 1. \]

Examples:
A Partial Order on States

The substate- (or child-) relation induces a partial order on states:

- top ≤ s, for all s ∈ S,
- s ≤ s′, for all s′ ∈ child(s),
- transitive, reflexive, antisymmetric,
- s′ ≤ s and s″ ≤ s implies s′ ≤ s″ or s″ ≤ s′.
Least Common Ancestor and Ting

- The least common ancestor is the function \( \text{lca} : 2^S \to S \) such that
  - The states in \( S_1 \) are (transitive) children of \( \text{lca}(S_1) \), i.e.
    \[ \text{lca}(S_1) \leq s, \text{ for all } s \in S_1 \subseteq S, \]
  - \( \text{lca}(S_1) \) is minimal, i.e. if \( \hat{s} \leq s \) for all \( s \in S_1 \), then \( \hat{s} \leq \text{lca}(S_1) \)

- **Note**: \( \text{lca}(S_1) \) exists for all \( S_1 \subseteq S \) (last candidate: top).
Least Common Ancestor and Ting

- Two states \( s_1, s_2 \in S \) are called \textit{orthogonal}, denoted \( s_1 \perp s_2 \), if and only if
  - they are unordered, i.e. \( s_1 \not\leq s_2 \) and \( s_2 \not\leq s_1 \), and
  - they live in different regions of an AND-state, i.e.
    \[
    \exists s, \text{region}(s) = \{S_1, \ldots, S_n\}, 1 \leq i \neq j \leq n : s_1 \in \text{child}(S_i) \land s_2 \in \text{child}(S_j),
    \]
Least Common Ancestor and Ting

- A set of states $S_1 \subseteq S$ is called consistent, denoted by $\downarrow S_1$, if and only if for each $s, s' \in S_1$,
  - $s \leq s'$,
  - $s' \leq s$, or
  - $s \perp s'$.
Legal Transitions

A hierarchical state-machine \((S, \text{kind}, \text{region}, \rightarrow, \psi, \text{annot})\) is called well-formed if and only if for all transitions \(t \in \rightarrow\),

- source and destination are consistent, i.e. \(\downarrow \text{source}(t)\) and \(\downarrow \text{target}(t)\),
- source (and destination) states are pairwise unordered, i.e.
  - for all \(s, s' \in \text{source}(t) (\in \text{target}(t))\), \(s \perp s'\),
- the top state is neither source nor destination, i.e.
  - \(\text{top} \notin \text{source}(t) \cup \text{source}(t)\).
- Recall: final states are not sources of transitions.

Example:
The Depth of States

- \( \text{depth}(\text{top}) = 0 \),
- \( \text{depth}(s') = \text{depth}(s) + 1 \), for all \( s' \in \text{child}(s) \)

Example:

![Diagram showing the depth of states with nodes and edges representing the states and transitions.

[Diagram showing the depth of states with nodes and edges representing the states and transitions.]
Enabledness in Hierarchical State-Machines

- The **scope** ("set of possibly affected states") of a transition $t$ is the least common region of

  $$source(t) \cup target(t).$$

- Two transitions $t_1, t_2$ are called **consistent** if and only if their scopes are orthogonal (i.e. states in scopes pairwise orthogonal).
Enabledness in Hierarchical State-Machines

- The **scope** ("set of possibly affected states") of a transition \( t \) is the least common region of
  \[ \text{source}(t) \cup \text{target}(t). \]
- Two transitions \( t_1, t_2 \) are called **consistent** if and only if their scopes are orthogonal (i.e. states in scopes pairwise orthogonal).
- The **priority** of transition \( t \) is the depth of its innermost source state, i.e.
  \[ \text{prio}(t) := \max\{\text{depth}(s) \mid s \in \text{source}(t)\} \]

- A set of transitions \( T \subseteq \rightarrow \) is **enabled** in an object \( u \) if and only if
  - \( T \) is consistent,
  - \( T \) is maximal wrt. priority,
  - all transitions in \( T \) share the same trigger,
  - all guards are satisfied by \( \sigma(u) \), and
  - for all \( t \in T \), the source states are active, i.e.
    \[ \text{source}(t) \subseteq \sigma(u)(st) (\subseteq S). \]
Transitions in Hierarchical State-Machines

- Let $T$ be a set of transitions enabled in $u$.
- Then $(\sigma, \varepsilon) \xrightarrow{(cons, Snd)} (\sigma', \varepsilon')$ if
  - $\sigma'(u)(st)$ consists of the target states of $t$,
    i.e. for simple states the simple states themselves, for composite
    states the initial states,
  - $\sigma'$, $\varepsilon'$, cons, and Snd are the effect of firing each transition $t \in T$
    one by one, in any order, i.e. for each $t \in T$,
    - the exit transformer of all affected states, highest depth first,
    - the transformer of $t$,
    - the entry transformer of all affected states, lowest depth first.

→ adjust (2.), (3.), (5.) accordingly.

Entry/Do/Exit Actions, Internal Transitions
Entry/Do/Exit Actions

- In general, with each state \( s \in S \) there is associated
  - an entry, a do, and an exit action (default: skip)
  - a possibly empty set of trigger/action pairs called internal transitions,
    (default: empty). \( E_1, \ldots, E_n \in \mathcal{E} \), ‘entry’, ‘do’, ‘exit’ are reserved names!

- Recall: each action’s supposed to have a transformer. Here: \( t_{act_1^{entry}}, t_{act_1^{do}}, \ldots \)
- Taking the transition above then amounts to applying
  \[ t_{act_2^{entry}} \circ t_{act} \circ t_{act_1^{exit}} \]
  instead of only
  \[ t_{act} \]
  \( \leadsto \) adjust (2.), (3.) accordingly.
For internal transitions, taking the one for $E_1$, for instance, still amounts to taking only $t_{act_{E_1}}$.

Intuition: The state is neither left nor entered, so: no exit, no entry.

- adjust (2.) accordingly.

Note: internal transitions also start a run-to-completion step.

Note: the standard seems not to clarify whether internal transitions have priority over regular transitions with the same trigger at the same state.

Some code generators assume that internal transitions have priority!
Alternative View: Entry/Exit/Internal as Abbreviations

... as abbreviation for ...

That is: Entry/Internal/Exit don’t add expressive power to Core State Machines. If internal actions should have priority, \( s_1 \) can be embedded into an OR-state (see later).

Abbreviation may avoid confusion in context of hierarchical states (see later).
Do Actions

- **Intuition**: after entering a state, start its do-action.
- If the do-action terminates,
  - then the state is considered **completed**,
- otherwise,
  - if the state is left before termination, the do-action is stopped.

Recall the overall UML State Machine philosophy:

"An object is either idle or doing a run-to-completion step."

Now, what is it exactly while the do action is executing...?
The Concept of History, and Other Pseudo-States

History and Deep History: By Example

What happens on...

- $R_s$?
  - $s_0, s_2$
- $R_d$?
  - $s_0, s_3$
- $A, B, C, S, R_s$?
  - $s_0, s_1, s_2, s_3, susp, s_4$
- $A, B, S, R_d$?
  - $s_0, s_1, s_2, s_3, susp, s_4$
- $A, B, C, D, E, R_s$?
  - $s_0, s_1, s_2, s_3, susp, s_4$
- $A, B, C, D, R_d$?
  - $s_0, s_1, s_2, s_3, susp, s_4$
Junction and Choice

- Junction ("static conditional branch"):
  - good: abbreviation
  - unfolds to so many similar transitions with different guards, the unfolded transitions are then checked for enabledness
  - at best, start with trigger, branch into conditions, then apply actions

- Choice: ("dynamic conditional branch")

Note: not so sure about naming and symbols, e.g., I’d guessed it was just the other way round...
Junction and Choice

- Junction ("static conditional branch"):  
  
  - **good**: abbreviation  
  - unfolds to so many similar transitions with different guards, the unfolded transitions are then checked for enabledness  
  - at best, start with trigger, branch into conditions, then apply actions

- Choice: ("dynamic conditional branch")  
  
  - **evil**: may get stuck  
  - enters the transition **without knowing** whether there’s an enabled path  
  - at best, use “else” and convince yourself that it cannot get stuck  
  - maybe even better: **avoid**

Note: not so sure about naming and symbols, e.g., I’d guessed it was just the other way round...

Entry and Exit Point, Submachine State, Terminate

- Hierarchical states can be “folded” for readability.  
  (but: this can also hinder readability.)  
- Can even be taken from a different state-machine for re-use.
Entry and Exit Point, Submachine State, Terminate

- Hierarchical states can be “folded” for readability.
  (but: this can also hinder readability.)
- Can even be taken from a different state-machine for re-use. $S : s$

**Entry/exit points**

- Provide connection points for finer integration into the current level, than just via initial state.
- Semantically a bit tricky:
  - First the exit action of the exiting state,
  - then the actions of the transition,
  - then the entry actions of the entered state,
  - then action of the transition from the entry point to an internal state,
  - and then that internal state’s entry action.

**Terminate Pseudo-State**

- When a terminate pseudo-state is reached, the object taking the transition is immediately killed.
Deferred Events in State-Machines

Active and Passive Objects [?]
What about non-Active Objects?

Recall:

- We’re still working under the assumption that all classes in the class diagram (and thus all objects) are active.
- That is, each object has its own thread of control and is (if stable) at any time ready to process an event from the ether.

But the world doesn’t consist of only active objects.
For instance, in the crossing controller from the exercises we could wish to have the whole system live in one thread of control.

So we have to address questions like:

- Can we send events to a non-active object?
- And if so, when are these events processed?
- etc.
propose the following (orthogonal!) notions:

- A class (and thus the instances of this class) is either active or passive as declared in the class diagram.
  - An active object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
  - A passive object doesn’t.

- A class is either reactive or non-reactive.
  - A reactive class has a (non-trivial) state machine.
  - A non-reactive one hasn’t.
Active and Passive Objects: Nomenclature

[?] propose the following (orthogonal!) notions:

- A class (and thus the instances of this class) is either active or passive as declared in the class diagram.
- An active object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
- A passive object doesn’t.

- A class is either reactive or non-reactive.
  - A reactive class has a (non-trivial) state machine.
  - A non-reactive one hasn’t.

Which combinations do we understand?

<table>
<thead>
<tr>
<th></th>
<th>active</th>
<th>passive</th>
</tr>
</thead>
<tbody>
<tr>
<td>reactive</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>non-reactive</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Passive and Reactive

- So why don’t we understand passive/reactive?
- Assume passive objects $u_1$ and $u_2$, and active object $u$, and that there are events in the ether for all three.

Which of them (can) start a run-to-completion step...?
Do run-to-completion steps still interleave...?
Passive and Reactive

- So why don’t we understand passive/reactive?
- Assume passive objects \( u_1 \) and \( u_2 \), and active object \( u \), and that there are events in the ether for all three.

Which of them (can) start a run-to-completion step...?  
Do run-to-completion steps still interleave...?

Reasonable Approaches:
- **Avoid** — for instance, by
  - require that reactive implies active for model well-formedness.
  - requiring for model well-formedness that events are never sent to instances of non-reactive classes.
- **Explain** — here: (following [?])
  - Delegate all dispatching of events to the active objects.

Passive Reactive Classes

- Firstly, establish that each object \( u \) knows, via (implicit) link \( \text{itsAct} \), the active object \( u_{\text{act}} \) which is responsible for dispatching events to \( u \).
- If \( u \) is an instance of an active class, then \( u_{\text{act}} = u \).
Passive Reactive Classes

- Firstly, establish that each object \( u \) knows, via (implicit) link \( \text{itsAct} \), the active object \( u_{\text{act}} \) which is responsible for dispatching events to \( u \).
- If \( u \) is an instance of an active class, then \( u_{\text{act}} = u \).

Sending an event:
- Establish that of each signal we have a version \( E \in C_{0,1}, C \in \mathcal{C} \).
- Then \( n!E \) in \( u_1 : C_1 \) becomes:
  - Create an instance \( u_e \) of \( E \in C_2 \) and set \( u_e \)'s dest to \( u_d := \sigma(u_1)(n) \).
  - Send to \( u_a := \sigma(\sigma(u_1)(n))(\text{itsAct}) \), i.e., \( \varepsilon' = \varepsilon \oplus (u_a, u_e) \).

Dispatching an event:
- Observation: the ether only has events for active objects.
- Say \( u_e \) is ready in the ether for \( u_a \).
- Then \( u_a \) asks \( \sigma(u_e)(\text{dest}) = u_d \) to process \( u_e \) — and waits until completion of corresponding RTC.
- \( u_d \) may in particular discard event.
And What About Methods?

- In the current setting, the (local) state of objects is only modified by actions of transitions, which we abstract to transformers.
- In general, there are also methods.
- UML follows an approach to separate
  - the interface declaration from
  - the implementation.
In C++ lingo: distinguish declaration and definition of method.
**And What About Methods?**

- In the current setting, the (local) state of objects is only modified by actions of transitions, which we abstract to transformers.
- In general, there are also **methods**.
- UML follows an approach to separate
  - the **interface declaration** from
  - the **implementation**.
  In C++ lingo: distinguish **declaration** and **definition** of method.

- In UML, the former is called **behavioural feature** and can (roughly) be
  - a **call interface** \( f(\tau_1, \ldots, \tau_n) : \tau_1 \)
  - a **signal name** \( E \)

Note: The signal list is redundant as it can be looked up in the state machine of the class. But: certainly useful for documentation.

---

**Behavioural Features**

<table>
<thead>
<tr>
<th>( C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \xi_1 f(\tau_1, \ldots, \tau_{1.n}) : \tau_1 P_1 )</td>
</tr>
<tr>
<td>( \xi_2 F(\tau_{2.1}, \ldots, \tau_{2.n_2}) : \tau_2 P_2 )</td>
</tr>
<tr>
<td>( \langle \text{signal} \rangle E )</td>
</tr>
</tbody>
</table>

**Semantics:**
- The **implementation** of a behavioural feature can be provided by:
  - An **operation**.
- The class’ **state-machine** (“triggered operation”).
**Behavioural Features**

\[
\begin{array}{c|c}
\text{C} & \\
\hline
\xi: f(\tau_1, \ldots, \tau_n) : \tau P_i \\
\xi: F(\tau_2, \ldots, \tau_n) : \tau P_2 \\
\{\text{signal}\} E \\
\end{array}
\]

**Semantics:**

- The **implementation** of a behavioural feature can be provided by:
  - An **operation**.
    - In our setting, we simply assume a transformer like \( T_f \).
    - It is then, e.g. clear how to admit method calls as actions on transitions:
      function composition of transformers (clear but tedious: non-termination).
    - In a setting with Java as action language: operation is a method body.
  - The class’ **state-machine** ("triggered operation").

- **Calling** \( F \) with \( n_2 \) parameters for a stable instance of \( C \) creates an auxiliary event \( F \) and dispatches it (bypassing the ether).
- Transition actions may fill in the return value.
- On completion of the RTC step, the call returns.
- For a non-stable instance, the caller blocks until stability is reached again.
### Behavioural Features: Visibility and Properties

<table>
<thead>
<tr>
<th>$\xi$</th>
<th>$\eta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\xi_1 f(\tau_1, \ldots, \tau_{m_1}) : \tau_1 P_1$</td>
<td>$\xi_2 f(\tau_2, \ldots, \tau_{m_2}) : \tau_2 P_2$</td>
</tr>
</tbody>
</table>

**Visibility:**
- Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.

#### Useful properties:
- **Concurrency**
  - concurrent — is thread safe
  - guarded — some mechanism ensures/should ensure mutual exclusion
  - sequential — is not thread safe, users have to ensure mutual exclusion
- **isQuery** — doesn’t modify the state space (thus thread safe)

For simplicity, we leave the notion of steps untouched, we construct our semantics around state machines. Yet we could explain pre/post in OCL (if we wanted to).
Discussion.

You are here.
References